DECENTRALIZED ALLOCATION OF SAFETY-CRITICAL APPLICATIONS ON PARALLEL COMPUTING ARCHITECTURE

A Dissertation
Presented to
The Academic Faculty

By

Louis Sutter

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Aerospace Engineering

Georgia Institute of Technology

December 2019

DECENTRALIZED ALLOCATION OF SAFETY-CRITICAL APPLICATIONS ON PARALLEL COMPUTING ARCHITECTURE

Approved by:

Dr. Eric Feron, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Glenn Lightsey
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Graeme J. Kennedy
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. S. K. Nandy
Department of Computational and Data Sciences
Indian Institute of Science

Date Approved: August 1st, 2019

To my family.

ACKNOWLEDGEMENTS

I would like to start by thanking my advisor, Prof. Eric Feron, first for offering me the opportunity to work on this research project, and also for his support throughout the completion of this thesis. His advice and his encouragement helped me a lot during my work.

I would also like to thank Prof. Lightsey, Prof. Kennedy and Prof. Chatterjee from the Georgia Institute of Technology, and Prof. Nandy, from the Indian Institute of Science, for accepting to be part of my thesis committee and for helping me to improve this work.

I would like to express my gratitude to Safran for sponsoring my research, and more specifically to Philippe Baufreton and François Neumann for their advice in order to produce a work that is relevant for the industry. I also thank Morphing Machines for sharing the details of the architecture they developed and that inspired this work.

I want to gratefully thank Thanakorn Khamvilai for his tremendous help during the design and the assembly of the experimental setting described in this thesis. I am also grateful to Tom Guillaumet, who laid the foundations of this work.

I also thank all my friends at the Georgia Institute of Technology for their support during my time in Atlanta.

Last but not least, I would like to warmly thank my whole family for their support and their encouragement during my studies far from them.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This work presents a decentralized task allocation algorithm for an abstract parallel computing architecture made of a set of Computational Units connected together and forming a network, each of them being prone to fail. Such an architecture can represent for example a multi-core processor with each Computational Unit standing for one core. The aim of the algorithm is to find the best mapping between Computational Units and the different applications we want to execute on the architecture, one of them being safety-critical and requiring continuity of service.

The proposed approach consists in formulating the allocation problem as an optimization problem. The architecture is represented by an abstract graph where each vertex stands for a Computational Unit and each edge stands for a communication link between two Computational Units. The applications are ranked by priority, the safety-critical application having the highest priority. Each application is also represented by the graph of the Computational Units they require for execution. From this, the allocation problem can be formulated as an Integer Linear Program (ILP), that is solved thanks to a state-of-the-art ILP solver. The objective function combines the maximization of the number of running applications, the minimization of reallocations since they interrupt the execution of the application, and the minimization of the length of communication paths between some applications. The constraints of the optimization problem take in particular into account the faulty Computational Units that must not be allocated to any application, and the requirement that a Computational Unit is dedicated to at most one application.

The second main aspect of this work is the decentralization the allocation process, in the sense that no central element decides alone of the allocation for the rest of the architecture. Redundant copies of the allocation algorithm are used instead, and they are executed on the

architecture itself, meaning that the copies must reallocate themselves. The combination of redundancy and majority voting systems allows the implementation of such a decentralized allocation process.

The proposed allocation process is then implemented on an experimental setup reproducing the REDEFINE architecture that inspired this work. REDEFINE is a multi-core processor in development at the Indian Institute of Science and the company Morphing Machines. In the presented experimental setup, each core is represented by a Raspberry Pi single board computer, forming a network imitating the Network-on-Chip of REDEFINE. The model is used to demonstrate the capabilities of the proposed allocation process to maintain operation of a physical system in a decentralized way while individual components fail.

CHAPTER 1
INTRODUCTION

1.1 Onset of multi-core processors

The onset of multi-core processors appeared as a golden opportunity for the embedded systems industry to improve efficiency of embedded computers. Multicore processors carry several benefits over single core ones, bringing more computational power through parallelization without increasing chip's internal frequency, and without increased energy consumption or increased heating. They now pervade cellular communication devices and embedded electronics for mass-market, for example, and many other industries are now taking advantage of such processors, such as the automotive industry [1], the biotechnology industry [2] and the circuit industry [3]. However, as far as critical systems are concerned, these benefits come with great certification challenges [4] [5], since parallel applications on a multi-core processor may interfere. The aerospace industry is yet undertaking to take up this challenge. In collaboration with the French aerospace company Safran, the Indian Institute of Science along with Morphing Machines developed REDEFINE[1] [6], a reconfigurable multi-core architecture that could host safety critical applications. REDEFINE can become an example of a safe multi-core processor by taking advantage of the inherent redundancy of such processors that enables graceful degradation [7]: when some cores fail, we can use the multiple remaining ones by reallocating affected applications to a healthy area of the chip.

[1]REDEFINE is a registered trademark of Morphing Machines.

1.2 Attempts to improve reliability

The inherent redundancy in such parallel architecture can also be seen as an opportunity to increase the reliability of computing systems, be it in safety critical embedded systems or for computing centers requiring guaranties of continuity of service.

For example, several attempts have been made to increase the reliability of safety-critical systems using multi-core processors.

In [8], an "hypervisor" is used to organize access to shared resources for applications, including safety-critical ones. However, a failure of this hypervisor is not taken into account in this patent. Therefore, such technique just moves the problem since the whole reliability is carried by the reallocation decision organ, which constitutes a single point of failure: the most complex and efficient reallocator is pointless if the system it executes on fails.

In [9], backup allocations are pre-calculated for each failure case and they are stored by individual Computational Units. For small architecture with only a few Computational Units, this solution is satisfactory and ensures a continuous fault tolerance of the system without requiring a centralized allocator. However, storing backup configurations can require a lot of memory when the architecture becomes bigger. Also, the proposed approach does not consider application that can themselves be parallelized and executed on several Computational Units at the same time.

Our approach differs from these two solutions by providing an on-line and decentralized reallocation algorithm for a general architecture that can be represented by a graph and for parallelized applications requiring several Computational Units to execute.

1.3 Objectives of the thesis

The objective of this thesis is to design a decentralized task allocation process for parallel computing architecture that can undergo faults. The algorithm is not specifically designed for a given architecture but can be applied to any parallel computing architecture that must execute a set of tasks.

This work presents a decentralized task allocation algorithm for an abstract parallel computing architecture made of a set of Computational Units (CUs) connected together and forming a network. Such an architecture can represent for example a multi-core processor, like the REDEFINE architecture that inspired this work, with each Computational Unit standing for one core. The aim of the algorithm is to find the optimal allocation of an a priori defined set of tasks on the architecture while taking into account the faults affecting the Computational Units (CUs). The faults are assumed to be detected by the algorithm when they occur, but this work does not provide a fault detection mechanism. As described later, two types of fault will be considered, the first one completely stopping the operation of the CU, and a second one considered to modify the computed output of the CU.

The second main feature of this work is the decentralized aspect of the allocation process. Decentralized means here that there is no central element deciding alone of the allocation for the rest of the architecture. Instead, we use redundant copies of the allocation algorithm executed on the architecture itself, meaning that the copies must reallocate themselves. This is achieved by using majority voting systems.

This work also presents an experimental setup reproducing several aspects of the REDEFINE architecture and used to implement the proposed decentralized allocation algorithm. The setup uses a network of Raspberry Pi single board computers to represent the cores of the REDEFINE architecture.

1.4 Thesis outline

This thesis is organized in 4 chapters. This first chapter consisted in a description of the context of this project and the motivations of this thesis.

The second chapter gives the theoretical elements describing the allocation problem, the method used to solve it and the approach for a decentralization of the allocation process.

Chapter 3 gives an application of the elaborated allocation process to a concrete example, a model of the multi-core processor REDEFINE, in development at the company Morphing Machines and the Indian Institute of Science (IISc).

Chapter 4 gives the conclusion of this work and provides thoughts for future work.

CHAPTER 2
THEORETICAL ASPECT

2.1 Mathematical description of the allocation problem

This section describes the mathematical formulation of the general allocation problem that is considered in this work. The idea is to use this mathematical formulation in an Integer Linear Program (ILP), whose solution is the best allocation of the tasks on the parallel computing platform (multi-core processors, network of computers in a computing center, etc), according to criteria described in Section 2.2.3, and taking into account the number of applications running, their priority, the number of reallocated applications and the length of communication paths between allocators and other applications.

The considered parallel computing platform is represented by a directed simple graph $\mathcal{G} = (V, E)$, where V is the set of *vertices* and $E \subseteq \{(x, y) \in V^2 \mid x \neq y\}$ is the set of *edges*. Each vertex of $\mathcal{G}$ represents a computational unit (CU), for example one core in a multi-core processor or at a different scale, one computer in a massively parallel supercomputer, and each edge of $\mathcal{G}$ represents a physical communication link between two computational units. The communication links are considered bidirectional, and therefore the orientation of edges can be chosen arbitrarily: we choose them to be oriented only to write more conveniently further constraints on the communication flow.

The graph $\mathcal{G}$ therefore represents the topology of the architecture. For example, the architecture can have a simple square mesh topology, as represented in Fig. 2.1.

From $\mathcal{G}$, we define parameters that will be used later in this work.

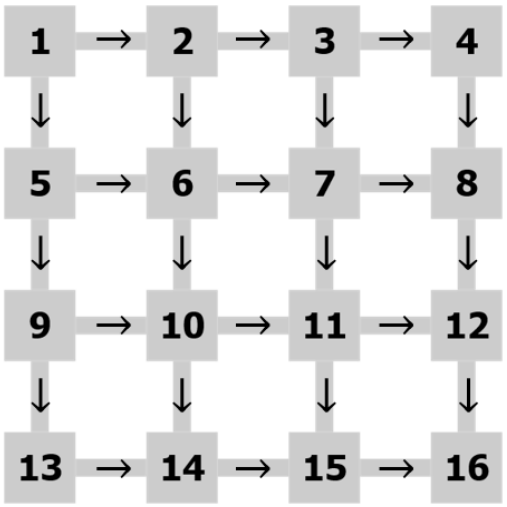


Figure 2.1: Example of square mesh topology. Orientation of edges are arbitrary.

Definition 1. N_{CUs} is defined as the number of Computational Units (CUs) in the computing platform, that is the number of vertices of $\mathcal{G}$.

Definition 2. N_{paths} is defined as the number of Physical Communication Links, or physical paths, in the platform, that is the number of edges of $\mathcal{G}$.

Let $N_{\text{app}} \in \mathbb{N}$ and $\mathcal{A} = \{\text{app}_k,\ k \in [\![1, N_{\text{app}}]\!]\}$ be a set of applications to be executed on the parallel computing platform. The applications in $\mathcal{A}$ are ranked by priority, app_1 having the highest priority and $\text{app}_{N_{\text{app}}}$ having the lowest one. The ranking is established *a priori* and represents the tolerated order in which we stop applications in case of computing resource failures. In the context of a commercial aircraft, an example of such applications with different priority would be the engine controller, with the highest priority, and a health monitoring application, with a lower priority, which is in charge of analyzing data from the engine in order to estimate its wear and to predict when maintenance operations are required. In case of computing resource failures, it would be tolerated in this context to stop the health monitoring application in order to maintain the execution of the engine controller.

For $k \in [\![1, N_{\text{app}}]\!]$, we assume that the compiler for the considered architecture de-

composes the application app_k into a undirected simple graph $\mathcal{G}_k = (V_k, E_k)$, where each vertex, that we will call *Application Node*, represents a sub-task of app_k that must be executed by a CU, and each edge represents a required communication link between two Application Nodes, that we will call an *Application Link*. Fig. 2.2 gives an example of such application graphs.

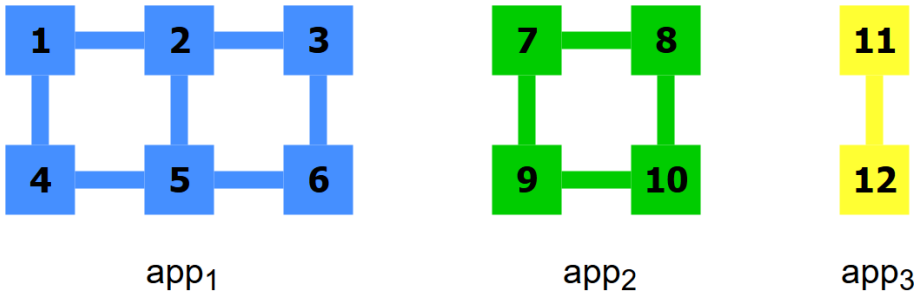


Figure 2.2: Example of application graphs. Each application node is identified with a unique index. app_1 has highest priority, app_3 has the lowest.

From each graph $\mathcal{G}_k$ for $k \in [\![1, N_{\text{app}}]\!]$, we define the following parameters.

Definition 3. N^k_{nodes} is defined as the number of Application Nodes in application k and N^k_{links} is defined as the number of Application Links in application k.

Definition 4. $N_{\text{nodes}} \coloneqq \sum_{k=1}^{N_{\text{apps}}} N^k_{\text{nodes}}$ is the total number of Application Nodes, and $N_{\text{links}} \coloneqq \sum_{k=1}^{N_{\text{apps}}} N^k_{\text{links}}$ is the total number of Application Links.

Each application node is given a global index $j \in [\![1, N_{\text{nodes}}]\!]$ with the following procedure: the nodes of app_1 keep the same indices as in the local numbering of vertices in $\mathcal{G}_1$; then the global indices for nodes of app_2 are obtained by increasing their local indices by N^1_{nodes} ; and so on for the nodes of app_k, by increasing the local numbering by $\sum_{l=1}^{k-1} N^l_{\text{nodes}}$. The result of the global numbering of the nodes can be seen on Fig. 2.2. An identical process is applied to obtain a global numbering of the edges of the application graphs.

The problem that we tackle here is to assign applications to CUs of the architecture while faults affect some CUs, taking into account the priority of the applications and specific constraints of the architecture. A solution will look like Fig. 2.3. The approach that we take here to solve the problem is to formulate the allocation problem as Integer Linear Program (ILP) and use a state-of-the-art ILP solver such as "GNU Linear Programming Kit" (GLPK) [10].

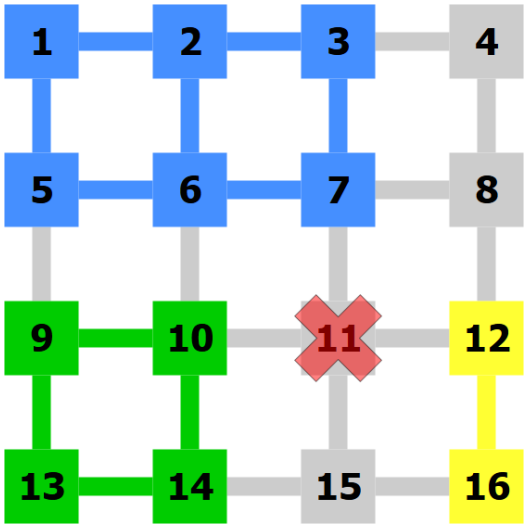


Figure 2.3: Example of a solution with a fault on core 11.

An additional aspect of the problem that we propose to solve is to make the allocation process decentralized, in the sense detailed in the introduction and in Section 2.3, with no central computing element allocating the tasks according to the solution of the ILP problem. The way this decentralized allocation is achieved is specifically described in Section 2.3: it involves several copies of the task computing the allocation and being executed on the platform itself. The number of such copies is the last parameter of our problem.

Definition 5. N_{realloc} is defined as the number of copies of the *Allocator Application*.

The next section details how the allocation problem is formulated as an ILP problem.

2.2 ILP formulation of the task allocation problem

2.2.1 Matrix representation of graphs

Definition 6. From the graph representation $\mathcal{G} = (V, E)$ of the parallel computing architecture, the $N_{\text{CUs}} \times N_{\text{paths}}$ ***NoC incidence matrix*** G associated with $\mathcal{G}$ is defined as:

$$[G]_{ij} \coloneqq \begin{cases} -1 & \text{if } e_j \in E \text{ leaves } v_i \in V, \\ 1 & \text{if } e_j \in E \text{ enters } v_i \in V, \\ 0 & \text{otherwise.} \end{cases}$$

And the $N_{\text{CUs}} \times N_{\text{paths}}$ ***NoC unoriented incidence matrix*** $\hat{G}$ associated with $\mathcal{G}$ is defined as:

$$[\hat{G}]_{ij} \coloneqq |[G]_{ij}| \,.$$

Definition 7. From the graph $\mathcal{G}_k = (V_k, E_k)$ representing the k-th application, the $N^k_{\text{nodes}} \times N^k_{\text{links}}$ ***application unoriented incidence matrix*** H^k associated with $\mathcal{G}_k$ is defined as:

$$[H]^k_{ij} \coloneqq \begin{cases} 1 & \text{if } v^k_i \in V_k \text{ and } e^k_j \in E_k \text{ are incident,} \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, the $N_{\text{nodes}} \times N_{\text{links}}$ ***overall application unoriented incidence diagonal block-matrix*** H is defined as

$$H \coloneqq \begin{bmatrix} H^1 & & \\ & \ddots & \\ & & H^k \end{bmatrix}.$$

2.2.2 Definition of the decision variables

Definition 8. The $N_{\text{CUs}} \times N_{\text{nodes}}$ *decision matrix* $X^{\text{CUs}\to\text{nodes}}$, mapping Application Nodes to CUs, is defined as:

$$X_{ij}^{\text{CUs}\to\text{nodes}} = \begin{cases} 1 & \text{if the CU } i \text{ is allocated to the Application Node } j, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 9. The $N_{\text{paths}} \times N_{\text{links}}$ *decision matrix* $X^{\text{paths}\to\text{links}}$, mapping Application Links to Physical Links, is defined as:

$$X_{ij}^{\text{paths}\to\text{links}} = \begin{cases} 1 & \text{if the Physical Link } i \text{ is allocated to the Application Link } j, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 10. The $N_{\text{apps}} \times 1$ *decision vector* r, representing which applications are executed, is defined as:

$$r_i = \begin{cases} 1 & \text{if the application } i \text{ is running,} \\ 0 & \text{if it is dropped.} \end{cases}$$

Definition 11. The $N_{\text{nodes}} \times 1$ *decision vector* M, representing which application nodes are reallocated, is defined as:

$$M_i = \begin{cases} 1 & \text{if the Application Node } i \text{ is moved from its previously allocated CU,} \\ 0 & \text{otherwise.} \end{cases}$$

Definition 12. For $k = 1, \ldots, N_{\text{realloc}}$, the $N_{\text{paths}} \times N_{\text{CUs}}$ *decision matrix* $X^{\text{Comm},\,k}$, repre-

senting communication paths between the k-th allocator application and every CU of the architecture, is defined as:

$$X_{ij}^{\text{Comm},\,k} = \begin{cases} -1 & \text{if the Physical Link } i \text{ is used to communicate between the allocator } k \text{ and the CU } j \text{ in the negative direction,} \\ 1 & \text{if the Physical Link } i \text{ is used to communicate between the allocator } k \text{ and the CU } j \text{ in the positive direction,} \\ 0 & \text{otherwise.} \end{cases}$$

Positive (respectively negative) direction means that the communication takes place in the same (respectively opposite) direction as the edge of the directed graph $\mathcal{G}$, as seen in Fig. 2.1.

2.2.3 Formulation of the optimization model

This section gives the detail of the formulation of the optimization problem that is solved each time a new fault is detected. This formulation includes the detail of the chosen objective function and constraints.

General form of the optimization model

The allocation problem is formulated as an Integer Linear Program (ILP) of the form:

$$\begin{aligned} &\text{maximize} && f(\boldsymbol{x}) = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ &\text{subject to} && M_1\boldsymbol{x} \leq \boldsymbol{b_1} \\ & && M_2\boldsymbol{x} = \boldsymbol{b_2} \\ &\text{and} && \boldsymbol{x} \text{ is a vector of integers.} \end{aligned} \tag{2.1}$$

$\boldsymbol{x}$ is the global vector of decision variables derived from the vectorization and the aggregation of the decision matrices from Section 2.2.2. We define it formally as:

$$\boldsymbol{x} = \begin{pmatrix} \mathrm{vec}(X^{\text{CUs}\rightarrow\text{nodes}}) \\ \mathrm{vec}(X^{\text{paths}\rightarrow\text{links}}) \\ r \\ M \\ \mathrm{vec}(X^{\text{Comm, }1}) \\ \vdots \\ \mathrm{vec}(X^{\text{Comm, }N_{\text{realloc}}}) \end{pmatrix}, \tag{2.2}$$

where vec is the common vectorization function for matrices: $\forall\, Q = (q_{i,j})_{1\leq i\leq m,\, 1\leq j\leq n}$, $\mathrm{vec}(Q) = [q_{1,1}, \ldots, q_{m,1}, q_{1,2}, \ldots, q_{m,2}, \ldots, q_{1,n}, \ldots, q_{m,n}]^{\mathrm{T}}$.

$\boldsymbol{c}$ is the coefficients of the objective function and M_1, M_2, $\boldsymbol{b_1}$ and $\boldsymbol{b_2}$ are parameters derived from the aggregation of the constraints of the problem that are described in the following sections. For example, for each scalar inequality constraint, after arranging the inequality with all decision variables on the left-hand side in the same order as in $\boldsymbol{x}$ and constant terms on the right-hand side, a row containing the coefficients of the decision vari-

ables is added to M_1 and the constant term is added in the vector $\boldsymbol{b_1}$. The same is done for equality constraints to build M_2 and $\boldsymbol{b_2}$.

Objective function

Given the priority of the applications in an ascending order, i.e. the first application has the highest priority and the N_{apps}-th application has the lowest one, the objective function is used in order to maximize the number of executed applications while minimizing the number of reallocations and the length of communication paths. The chosen objective function, in terms of $\boldsymbol{x}$ as defined above in equation 2.2, is:

$$\max \left\{ f(\boldsymbol{x}) = \sum_{k=1}^{N_{\text{apps}}} \alpha_k \cdot r_k - (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} M_j - \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} \left| X_{ij}^{\text{Comm},\, k} \right| \right\}, \tag{2.3}$$

where

$$\begin{aligned} \beta &= N_{\text{realloc}} \times N_{\text{CUs}} \times N_{\text{paths}}, \\ \alpha_{N_{\text{apps}}} &= (\beta + 1) \times N_{\text{nodes}} + \beta + 1, \\ \text{and } \forall k < N_{\text{apps}} &: \\ \alpha_k &= \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + (\beta + 1) \times N_{\text{nodes}} + \beta + 1. \end{aligned} \tag{2.4}$$

The coefficients of the objective function are chosen to prioritize the different aspects that are optimized in this function:

1. The first priority is to execute each application, even if it means more reallocations and longer communication paths.

2. Then, minimizing the number of reallocations is more important than having shorter communication paths, since a reallocation temporarily interrupts the execution of the allocation.

3. When running all applications is not feasible, the priorities of the applications are enforced and executing any given application is more important than running any number of applications with a lower priority. However, if because of its geometry, a given application cannot be executed anyway, nothing prevents lower-priority applications from being executed.

These requirements motivated the choice for the coefficients in the objective function. The proof that these coefficients allow the objective function to meet these requirements is given in Appendix A.

Note that the problem of minimizing or maximizing the absolute value of the $X_{ij}^{\text{Comm},\,k}$ variables, which is a nonlinear program, can be reformulated as a linear program by introducing additional variables and constraints [11], that were not presented in the previous section for conciseness. For each entry $X_{ij}^{\text{Comm},\,k}$ of X^{Comm}, an auxiliary variable $\hat{X}_{ij}^{\text{Comm},\,k}$ is introduced to represent its absolute value, and two extra constraints are added:

$$+X_{ij}^{\text{Comm},\,k} \leq \hat{X}_{ij}^{\text{Comm},\,k},$$

$$-X_{ij}^{\text{Comm},\,k} \leq \hat{X}_{ij}^{\text{Comm},\,k}.$$

$\hat{X}_{ij}^{\text{Comm},\,k}$ is then used instead of $\left|X_{ij}^{\text{Comm},\,k}\right|$ in the objective function. Because the objective function tends to maximize $-\left|X_{ij}^{\text{Comm},\,k}\right|$, so to minimize $\hat{X}_{ij}^{\text{Comm},\,k}$, one of the two previous constraints will be binding, the stricter one, where the left-hand side is the greatest and equal to $\max(+X_{ij}^{\text{Comm},\,k}, -X_{ij}^{\text{Comm},\,k})$, which is exactly $\left|X_{ij}^{\text{Comm},\,k}\right|$. The other constraint will be non-binding and therefore does not affect the optimal point. It thus ensures

that $\hat{X}_{ij}^{\text{Comm},\,k}$ is equal to $\left|X_{ij}^{\text{Comm},\,k}\right|$.

Constraints

Domain of decision variables The decision variables $X^{\text{CUs}\to\text{nodes}}$, $X^{\text{paths}\to\text{links}}$, r and M are binary i.e. the value of their entries must be either 0 or 1.

The entries of $X^{\text{Comm},\,k}$ for $k \in [\![1,\ N_{realloc}]\!]$ must belong to $\{-1,\ 0,\ 1\}$.

Resource allocation and partitioning Several equations express the constraints of allocating the resources of the Fabric to applications while enforcing partitioning on the chip.

- Each CU can be allocated to at most one application, as a way to enforce spatial partitioning of applications on the NoC, i.e.

$$\forall i = 1, \ldots, N_{\text{CUs}},\ \sum_{j=1}^{N_{\text{nodes}}} X_{ij}^{\text{CUs}\to\text{nodes}} \leq 1. \tag{2.5}$$

- Each running Application Node must be assigned to exactly one CU, i.e.

$$\forall i = 1, \ldots, N_{\text{nodes}},\ \sum_{j=1}^{N_{\text{CUs}}} X_{ji}^{\text{CUs}\to\text{nodes}} = r_{N(i)}. \tag{2.6}$$

  $N(i)$ is the application number corresponding to Application Node i.

- A physical communication link of the architecture can be allocated to at most one

Application Link[1], i.e.

$$\forall i = 1, \ldots, N_{\text{paths}},\ \sum_{j=1}^{N_{\text{links}}} X_{ij}^{\text{paths}\to\text{links}} \leq 1. \tag{2.7}$$

- Each running Application Link must be assigned to exactly one physical communication link of the architecture, i.e.

$$\forall i = 1, \ldots, N_{\text{links}},\ \sum_{j=1}^{N_{\text{paths}}} X_{ji}^{\text{paths}\to\text{links}} = r_{L(i)}. \tag{2.8}$$

$L(i)$ is the application number corresponding to Application Link i.

Compliance with the architecture An Application link, adjacent to an Application Node that has been mapped to a given CU, must be allocated to a Physical Link that is adjacent to that CU, i.e.

$$X^{\text{CUs}\to\text{nodes}}\ H = \hat{G}\ X^{\text{paths}\to\text{links}}. \tag{2.9}$$

This equation is equivalent to the scalar equations:

$$\forall i = 1, \ldots, N_{\text{CUs}},\ \forall j = 1, \ldots, N_{\text{links}},$$

$$\sum_{k=1}^{N_{\text{nodes}}} X_{ik}^{\text{CUs}\to\text{nodes}}\ H_{kj} = \sum_{l=1}^{N_{\text{paths}}} \hat{G}_{il}\ X_{lj}^{\text{paths}\to\text{links}}.$$

The left-hand side is equal to one if and only if the CU i has been allocated to Application Node k and Application Node k is adjacent to Application Link j. The right-hand side is equal to one if and only if the CU i is adjacent to the Physical Link l and the Physical Link l is allocated to Application Link j, which proves the correctness of the constraint.

[1]This does not mean that this communication link cannot be used for other communication purposes on the architecture, but only one of the Application Link computed by the compiler for the applications can be allocated to that physical communication link.

Link with the previous allocation A given Application Node can either remain affected to the same CU, either be moved or be dropped:

$$\begin{aligned} \forall i \in [\![1, N_{\text{CUs}}]\!],\ \forall j \in [\![1, N_{\text{nodes}}]\!],\ \text{s. t. } X_{\text{old } ij}^{\text{CUs}\to\text{nodes}} = 1, \\ \left(1 - r_{N(j)}\right) + M_j + X_{ij}^{\text{CUs}\to\text{nodes}} = X_{\text{old } ij}^{\text{CUs}\to\text{nodes}}, \end{aligned} \tag{2.10}$$

with $X_{\text{old}}^{\text{CUs}\to\text{apps}}$ be the parameter containing the mapping between CUs and Application Nodes computed during the previous allocation.

This constraint is ignored for the initial allocation.

Faults We assume the parallel computing architecture is equipped with a fault detection system that can detect and inform the allocators when a CU fails. From this information, we can add constraints to take into accounts fault in the architecture. Within a CU i:

- if the CU is healthy, any Application Node can be mapped on the CU,
- if the CU is faulty, then no Application Nodes can be mapped on the CU i:

$$\sum_{k=1}^{N_{\text{nodes}}} X_{ik}^{\text{CU}\to\text{apps}} = 0. \tag{2.11}$$

The detection of this fault is either assumed for the model or detected by the voter using the majority rule described in 3.2.2.

Communication constraints Since the allocators are executed on the platform, we must ensure that they will be able to send the allocation they computed to the other CUs of the architecture, given the communication links that allows each CU to send a message only through its neighbors. Therefore, we must make sure that there exists a path from each

allocator to the other CUs:

$$\forall k \in [\![1, N_{\text{realloc}}]\!],\ G\ X^{\text{Comm},\,k} = S^k, \tag{2.12}$$

where S^k is the $N_{\text{CUs}} \times N_{\text{CUs}}$ *source-sink matrix* S^k, depending on $X^{\text{CUs}\to\text{nodes}}$ and defined by:

$$[S]^k_{ij} \coloneqq \begin{cases} 0 & \text{if } \deg(v_i) = 0 \text{ in } \mathcal{G}, \\ 0 & \text{if CU } i \text{ is faulty}, \\ -X^{\text{CUs}\to\text{nodes}}_{i\ \text{node\_of\_alloc}(k)} + [I_{N_{\text{CUs}}}]_{ij} & \text{otherwise}, \end{cases}$$

where the degree of a vertex $\deg(v_i)$ is the number of edges connected to it, and node_of_alloc(k) is the Application Node corresponding to allocator k. When the CU is neither faulty nor without any neighbor, in each path between an allocator and another given CU i, the allocator is the source (-1) and the CU i is the sink (+1).

Constraints specific to the architecture Additional constraints can be added to comply with specific aspects of the considered architecture.

For example, we will describe the REDEFINE architecture in Section 3.1, where intra-application communication between CU can happen only in a specific way as described later. Because of this, orientation of the applications on the architecture matters, because nodes that can communicate in a given orientation will not be able to do so if they are rotated on the architecture. Therefore the orientation as computed by the compiler must be enforced.

To ensure correct orientation of applications, a set of constraints used in [12] is also required. In order to enforce this, the numbering of the CUs on the architecture is used. For example, as illustrated in Fig. 2.4, a CU has always a number difference of -1 with its

right neighbor and $+N_{\text{row}}$ with its top neighbor, where N_{row} is the number of Tiles per row of the NoC ($N_{\text{row}} = 4$ in our example).

The difference between the numbers of the contiguous pairs of CUs allocated to an application must match the orientation computed by the compiler.

Let j^k be the index of the top-left node of the k-th application:

$$\begin{aligned} &\forall i \in [\![1, N_{\text{CUs}}]\!],\ \forall k \in [\![1, N_{\text{apps}}]\!], \\ &X_{ij^k}^{\text{CUs}\to\text{nodes}} = X_{(i+1)(j^k+1)}^{\text{CUs}\to\text{nodes}} \\ &X_{ij^k}^{\text{CUs}\to\text{nodes}} = X_{(i+N_{\text{row}})(j^k+N_{\text{row}}^k)}^{\text{CUs}\to\text{nodes}}, \end{aligned} \tag{2.13}$$

where N_{row}^k is the number of nodes per row of the k-th application.

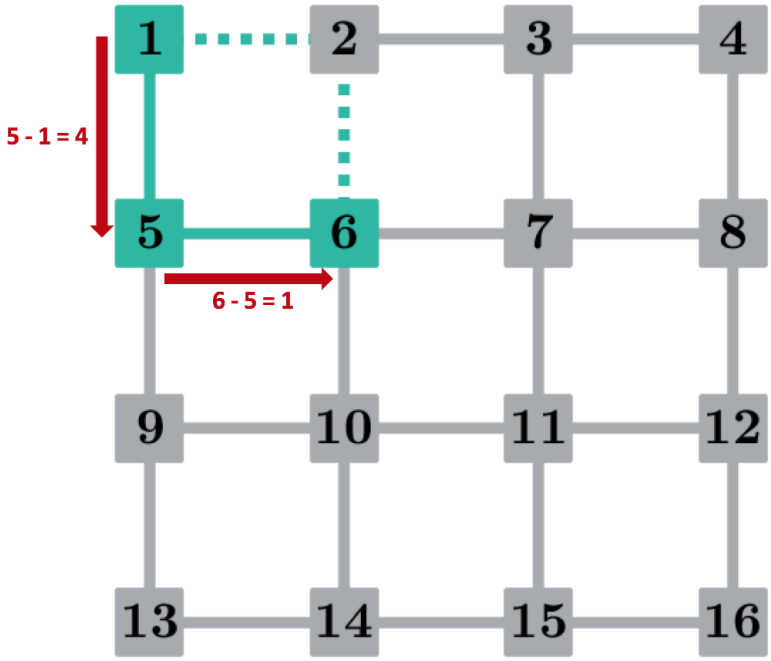


Figure 2.4: Enforcement of the spatial orientation of the application by equating the difference between CUs' indices allocated to it.

2.3 Decentralization of the allocation system

In this work, we use the word "decentralized" to qualify a system where no single computing element has control over all the other ones in the parallel architecture: there is no central computing element whose failure jeopardizes the operation of the whole parallel architecture. In safety words, this means that no computing element constitutes a single point of failure.

We focus here on computing elements, but there are other elements that may be a single point of failure and that we do not take into account in this work. For example, electrical power may be provided by one unique and central power supply unit, which is an obvious single point of failure if not designed carefully.

The proposed algorithm in [12] and [13] is executed on a unique Resource Manager, located outside of the Network-on-Chip (NoC). If this Resource Manager fails, the system loses all reallocation possibilities, which heavily degrades the reliability of the system. This motivates a decentralized implementation of the algorithm.

2.3.1 N-modular redundancy and majority voting system

To develop a decentralized allocation system for the considered parallel computing platform, we chose to use the concept of N-modular redundancy with a majority voting system [14].

In this approach, N_{realloc} is an odd number greater or equal to 3 and N_{realloc} copies of the same sub-tasks are executed in parallel. N_{realloc} is taken odd to avoid the case where equal number of copies agree on two different results. The copies are fed with the same inputs and their outputs are then sent to a majority voting system. As illustrated in Fig. 2.5,

the voting system compares the outputs of the redundant copies and filters them: only the result that has been computed by the majority of the redundant copies will be transmitted, i.e. the result computed by at least $\frac{N_{\text{realloc}}+1}{2}$ redundant copies. The voting system is also used to report the failure of the redundant copies that do not match the majority result.

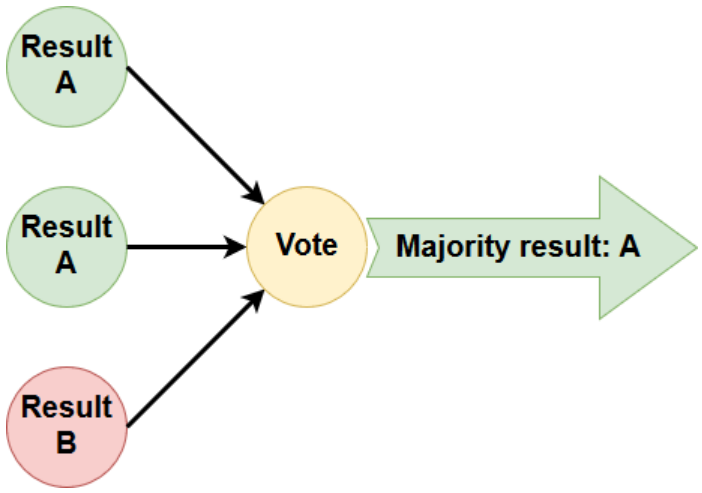


Figure 2.5: Illustration of the voting process with 3 redundant copies.

2.3.2 Decentralized implementation

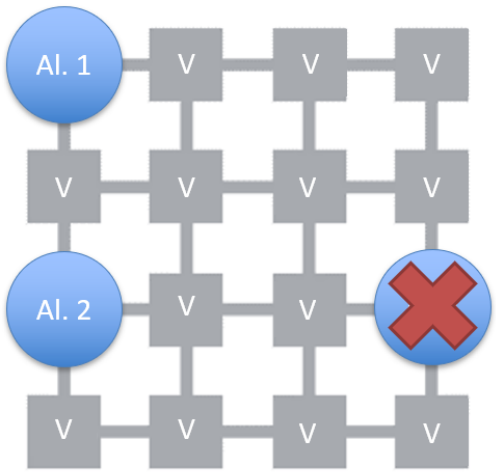
The proposed idea to decentralize the allocation system is to execute N_{realloc} modular redundant copies of the allocation task on the architecture itself, with a voting system implemented on each CU. For further examples, N_{realloc} will be taken equal to 3.

In normal conditions, the N_{realloc} copies of the allocator compute the same allocation, since they solve an identical ILP problem, with same inputs and constraints, and because GLPK is a deterministic solver. This allocation is then broadcast to every CU, including the ones executing the allocators.

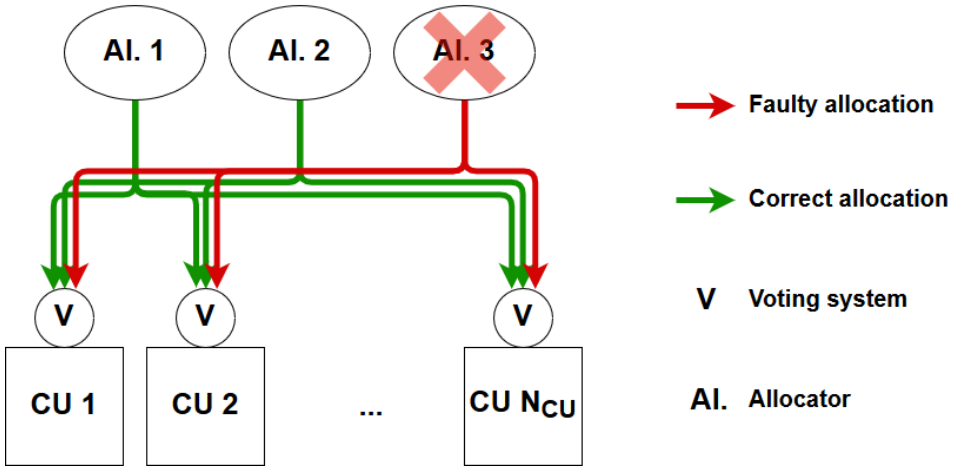
If a CU not running an allocator fails, all 3 allocators compute the same new allocation, in which the affected application is assigned to a new CU, according the algorithm

described previously in Section 2.2. This new allocation is then broadcast and received by all CUs. Since the 3 signals that the CUs receive are coherent, they all comply with it and therefore, the affected application is reallocated.

On the other hand, as illustrated in Fig. 2.6, if a CU that was running a copy of the allocator is affected by a fault, the 2 other ones will compute the same new allocation where the affected copy is assigned to a new healthy CU. Regardless of what the faulty allocator computes, only the two coherent allocation sent by the two healthy allocators will be taken into account by the CUs, and the faulty allocator will be reallocated.



(a) Layout of the allocators on the computing architecture.



(b) Information flow between allocators and CUs.
The correct allocation includes the instruction for some node i to run allocator 3.

Figure 2.6: Fault affecting a CU running an allocator.

CHAPTER 3
PRACTICAL EXAMPLE

3.1 The REDEFINE architecture

The REDEFINE many-core architecture [6] that inspired this work is an architecture in development at the company Morphing Machines and the Indian Institute of Science. Unlike other architectures, REDEFINE features a dynamic reconfiguration capability: applications running on the chip can be dynamically allocated to different cores of the processor. This feature motivated the development of an algorithm that computes the allocation of applications according to the state of degradation of the chip and the criticality of each application.

REDEFINE is made of two main elements: an Executable Fabric and a Resource Manager. It is connected to an external memory and a host, as shown in Fig. 3.1.

The Execution Fabric is a toroidal mesh of a certain number of Tiles connected through the Network on Chip (NoC), as shown in Fig. 3.2. Each Tile includes a router (shown as a pink circle) and a Computer Resource (shown as a gray box and denoted CR thereafter) that is responsible for actual computations. Some extra Tiles are added on one edge of the Fabric to ensure the communication with the Resource Manager. These "Gateway Tiles" support routing functions only.

The REDEFINE Resource Manager (RRM) is the interface between the Fabric and the host that creates the decomposition of applications into sequences of basic elements called HyperOps. As such, the RRM is in charge of allocating parts of the Fabric to the different HyperOps, launching them, getting the results through the Gateway Tiles, and transferring

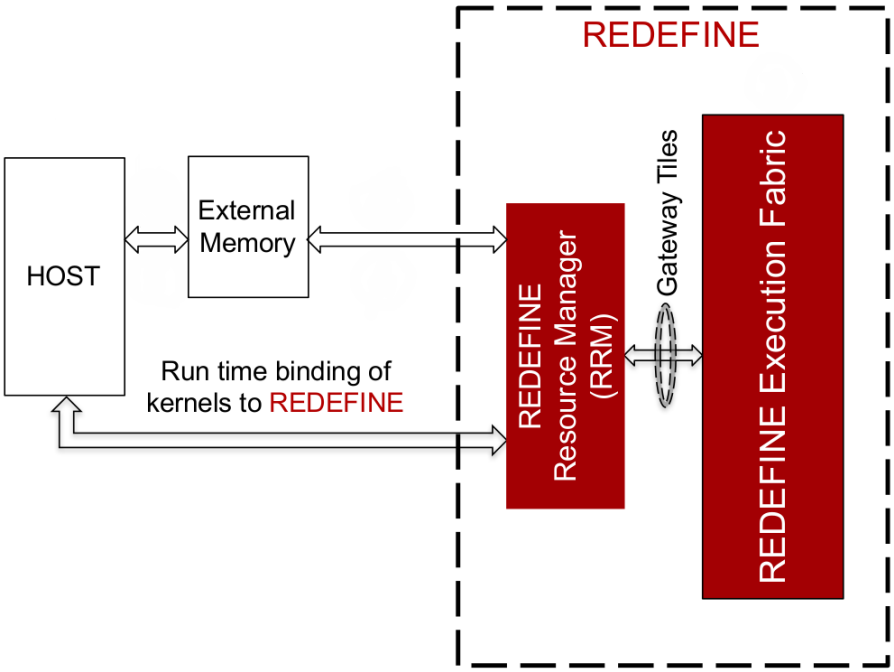


Figure 3.1: The different components of the REDEFINE architecture.

them to the host.

The sequences of HyperOps are generated from C code during the offline compilation process, which also computes their spatial configuration that must be respected on the REDEFINE Fabric (Fig. 3.3). Each HyperOp is designed to be executed by one REDEFINE Tile, the basic element of the Fabric. The necessity to respect the orientation of the HyperOps pattern comes from the XY deterministic routing algorithm [15] used on the Fabric: to reach its destination, a packet first moves horizontally along the X-axis to reach the destination's column and then vertically along the Y-axis to reach its row. HyperOps that need to communicate are therefore required to have a specific orientation.

An example of spatial configuration for an application computed by the compiler is given in Fig. 3.4. It gives the relative position of the HyperOps, also denoted "True Application Nodes" thereafter, that must be assigned to some Tiles of the Fabric. These True

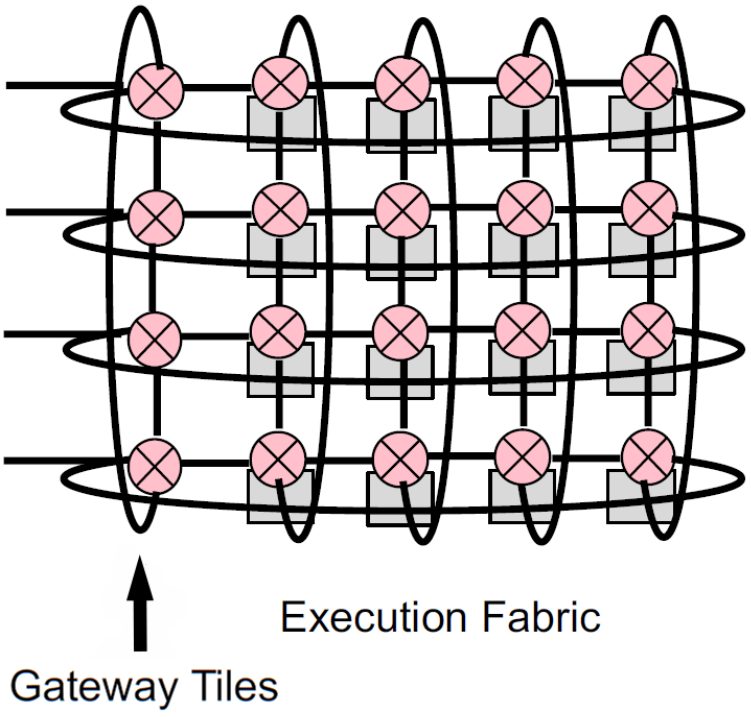


Figure 3.2: Example of a toroidal mesh topology of the NoC for a 4×4 fabric. The pink circles represent routers, and the gray squares represent Compute Resources.

Application Nodes therefore require the CR of those Tiles. The router of some extra Tiles may also be needed for intra-application communication between Application Nodes. To ensure spatial partitioning of the applications on the Fabric and to prevent another application from using them, these extra Tiles are considered as entirely allocated to the application as well: the term "Ghost" Application Nodes then denotes fictitious Nodes that must be assigned to such Tiles.

A simple model of faults is considered, in which each component of a Tile, the CR and the router, can be either functional or permanently faulty.

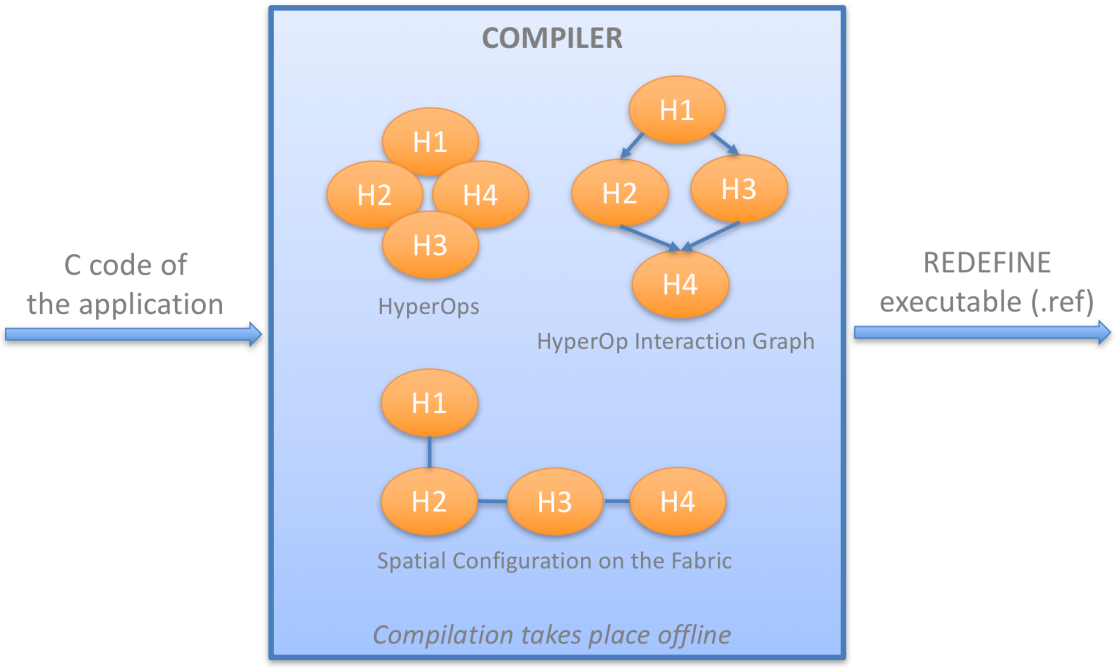


Figure 3.3: Illustration of the compilation process.

3.2 Raspberry Pi representation of REDEFINE

3.2.1 Hardware components

To illustrate and demonstrate the capabilities of the new formulation of the allocation algorithm in operational conditions, we choose to implement it on a macroscopic, simplified representation of the REDEFINE architecture, in order to control and maintain operation of a physical system despite the presence of faults.

Replication of REDEFINE

Since the REDEFINE architecture is still in development, we chose to build a hardware model of it. In this model, each Tile of the REDEFINE architecture is represented by a Raspberry Pi computer [16]. A REDEFINE Execution Fabric of 4×4 cores is thus replicated with a network of 16 Raspberry Pi computers. All of them are connected to a

Figure 3.4: Example of the spatial configuration of an application. Colored squares represent “True Nodes”, corresponding to a Tile whose CR must be allocated to the application. A “Ghost” Application Node (in gray) is added because the top right Tile’s router is used by the application for intra-application communication. Such a Node is considered to be part of the application.

common routing switch in a local area network (LAN). At this stage of the work, we did not try to accurately reproduce the communication protocol used on the REDEFINE Fabric [6]. Also, for simplicity of visualization, the network is considered to be a square mesh, instead of a toroidal mesh.

The goal of this model of REDEFINE is also to show the possibility to decentralize the allocation process. Therefore, no external Resource Manager is used. Instead, 3 copies of the allocator are executed on the network, as described in Section 2.2.

Faults

Two types of faults are considered in this experiment. The first type is computational fault, which randomly affect the computations performed by the Raspberry Pi. We detect this kind of fault by using redundant copies of the considered application combined with a voting system that is described below in section 3.2.2. The second type of fault is assumed to stop the operation of the Tile it affects. We also assume that this fault can be detected by the Resource Manager. In practice, each time one of these faults affects a Raspberry Pi, the status signal sent by this Raspberry Pi to the Resource Manager is changed to a

signal identifying it as faulty. Each of these two kinds of fault can be manually triggered or recovered thanks to a breadboard as seen in Fig. 3.5 connected to each Raspberry Pi.

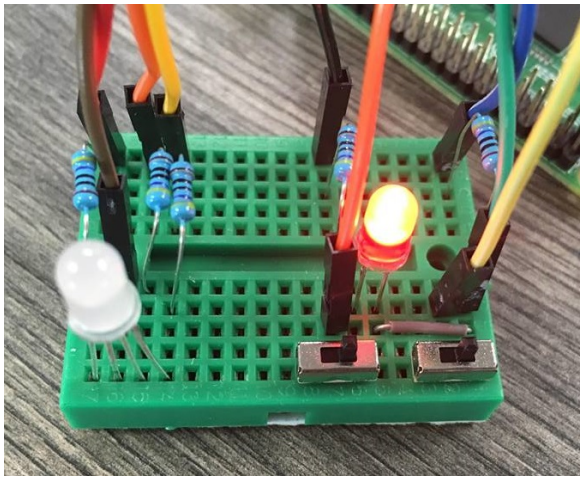


Figure 3.5: Hardware associated with each Raspberry Pi Tile.
The RGB-LED (bottom-left corner) indicates which application is executed. The red LED (right side) indicates an healthy Tile when turned on. Each switch is used to trigger one type of fault.

Controlled system

The physical system we chose to control with this model of the REDEFINE architecture is a propulsive system, made of an electric fan mounted on a thrust stand. The fan is commanded by using Pulse Width Modulation (PWM). The measure of the thrust is used by a simple proportional controller executed on the Fabric to compute the value of the PWM command required to maintain the thrust at a constant value.

An extra Raspberry Pi is used as the micro-controller of the fan: it converts the value measured by the load cell, sends it to the "Execution Fabric" where the appropriate control value is computed, and generates the corresponding PWM signal controlling the fan. It must therefore be noted that although the same hardware representation is used, this Raspberry Pi does not correspond to the same components as the ones used for the Tiles of the Execution Fabric.

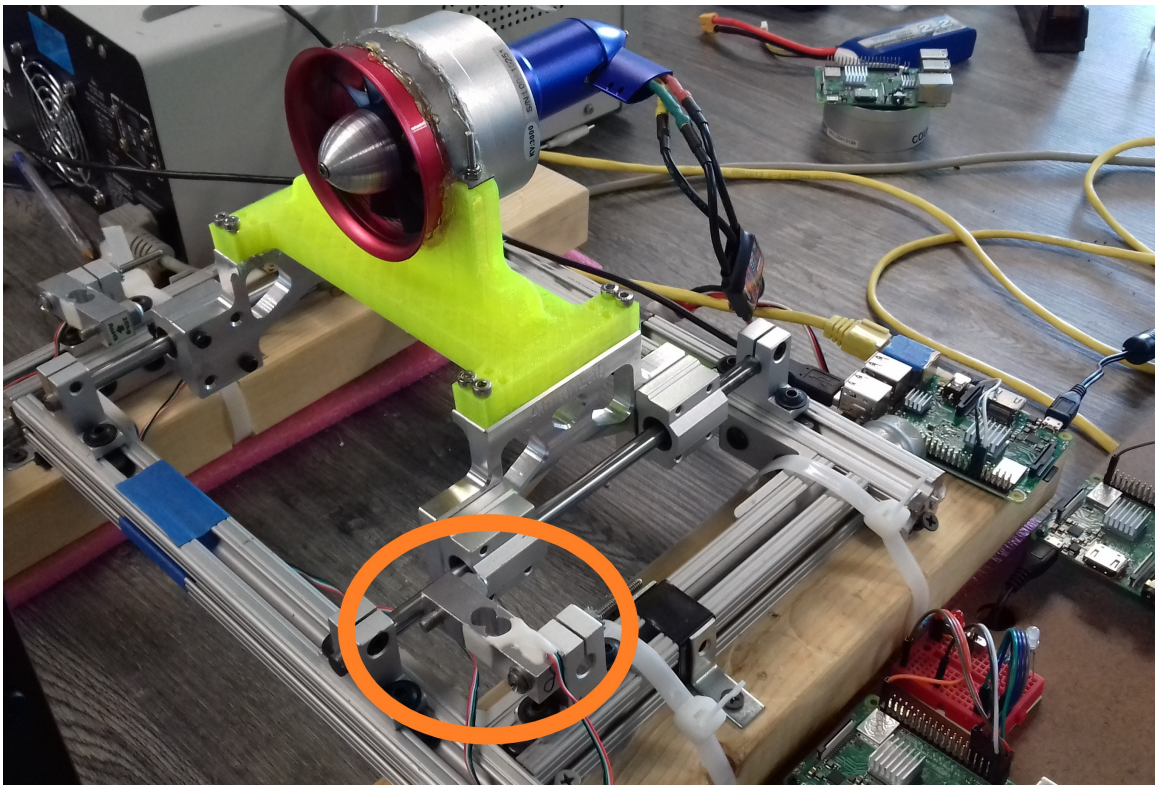


Figure 3.6: Electric fan mounted on the thrust stand.
The delivered thrust is measured thanks to a load cell on the stand, indicated by the orange circle.

3.2.2 Software components

Even if a controller is reallocated to healthy Tiles when it is affected by a fault, because of the time required to compute the new allocation and to actually reallocate the set of tasks, the operation of the fan may be temporarily altered during the reallocation process.

To avoid interruptions in the operation of the fan during reallocations, we also use a standard Triple Modular Redundancy (TMR) architecture [14]. Three copies of the controller are executed on the Fabric. Each one separately computes the duty-cycle value of the PWM signal that should be sent to the fan, given the thrust value that they all receive from the sensor. The three values are sent to the Raspberry Pi representing the micro-controller of the fan, where a voting system decides which control output should be used. The vote

outputs the result that has been computed by the majority of the controllers, in this case 2 out of 3. Signals are here considered equal if their difference is smaller than a given tolerance. In the case of a fault affecting the output of one of the controllers, the two remaining healthy controllers ensure that the correct value is sent to the fan. The voting system also identifies which controller is not coherent with the two others and informs the Resource Manager of the fault. The reallocation process that we implemented can then take place while providing continuity of service with the two healthy controllers. To complicate the reallocation tasks, each copy of the controller has been arbitrarily attributed to 2 Application Nodes. Concretely, only one of them is responsible of actual computations.

Three copies of the allocator execute the allocation algorithm itself. They have second rank priority immediately below the controllers, which represent the safety-critical application in this case. Giving the allocators only the second rank in the priority list can be justified when considering the case where only a controller or an allocator can be executed on the architecture: the resource must be allocated to the safety-critical application, in this case the controller, that maintains the operation of the system, whereas the allocator is only a protection against further faults, but cannot alone ensure operation of the controlled system.

In addition to these six applications, one dummy application is considered in this experiment: it occupies 2 Tiles of the Fabric, but does not perform actual computation except changing the voltage in the RGB LED to display its corresponding color. It has the lowest priority.

Figure 3.7 sums up the list of considered applications for the experiment, their relative priority and the resources they require in terms of number of Tiles. The initial allocation of these applications on the model is given in Fig. 3.8.

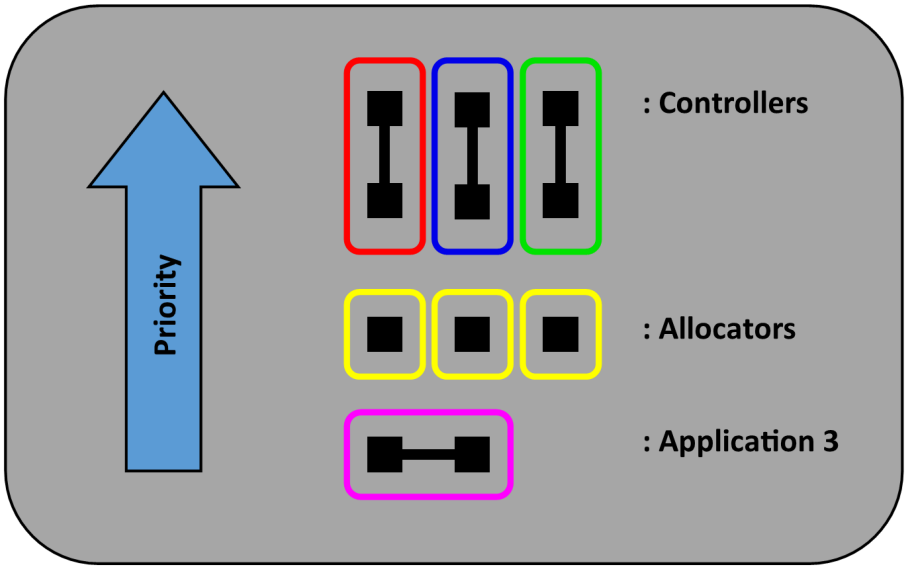


Figure 3.7: Considered applications for the experiment, their priority and the number of Tiles they require.

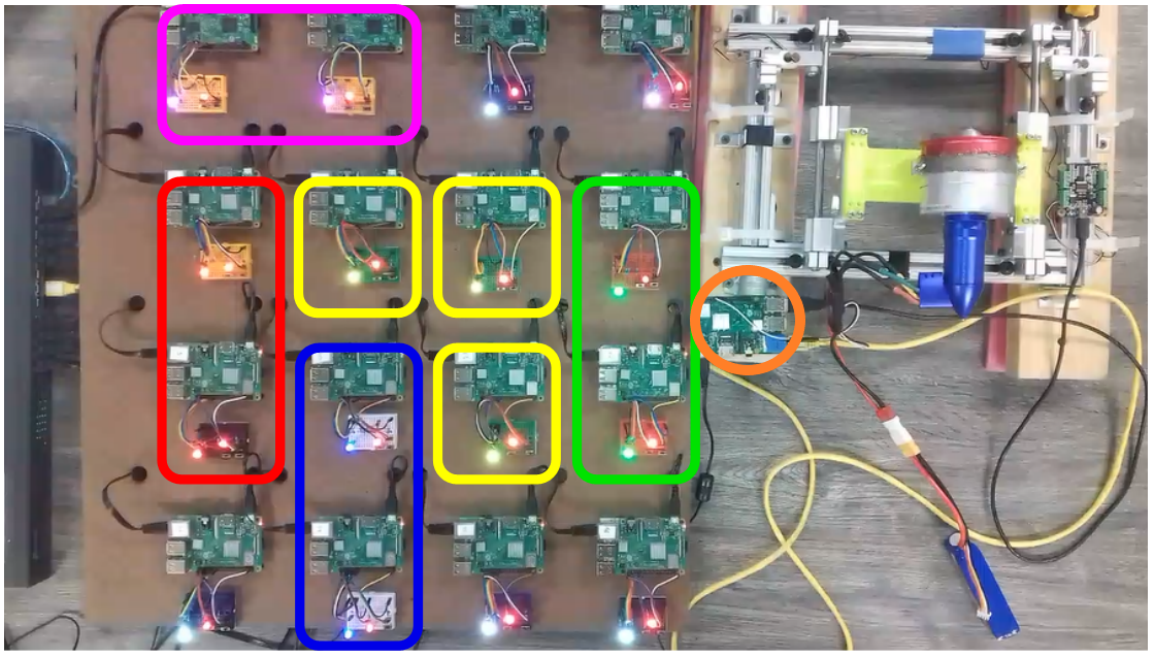


Figure 3.8: Initial allocation of the applications on the model.
The orange circle identifies the extra Raspberry Pi for interactions with the stand.

3.3 Results

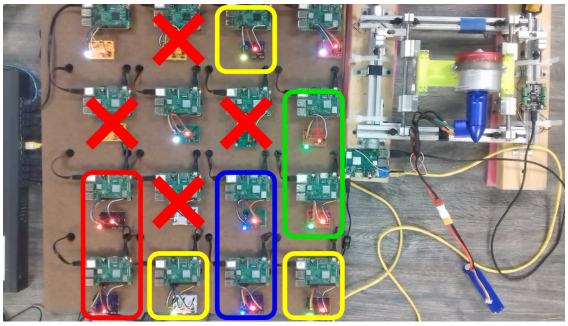
Starting from the initial allocation given in Fig. 3.9, faults are triggered on the model. After each fault, the allocators detect the faulty Raspberry Pi and compute a new allocation that is then broadcast on the network. They maintain the execution of the safety-critical application as long as enough resources are available for it.

Tiles surrounded by faulty neighbors are isolated from the rest of the architecture and cannot communicate. As enforced by the communication constraints described in paragraph 2.2.3, such a Tile is not given any task to execute and is as good as faulty, as seen in Fig. 3.9a.

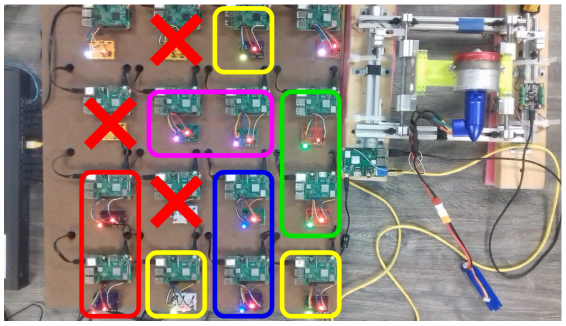
When a Tile recovers from a fault, an application can be allocated back to it as seen in Fig. 3.9b. Applications are dropped according to their priority when more Tiles become faulty. However, as illustrated in Fig. 3.9d, when no space is available for all first priority applications, lower priority ones are still allowed to be executed.

The voting system implemented on the fan needs at least two functioning and coherent controllers to run the fan (Fig. 3.9d), as previously explained in section 3.2.2: in case the signals received from the controllers are incoherent, it decides not to trust any of them and the engine stops, as in Fig. 3.9e.

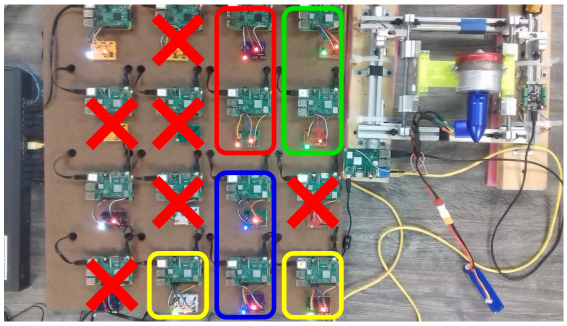
Since the controllers have the highest priority, they are the last remaining applications to be executed in Fig. 3.9g. After this step, further faults will affect the controllers but no reallocation can happen because no more allocator is executed.



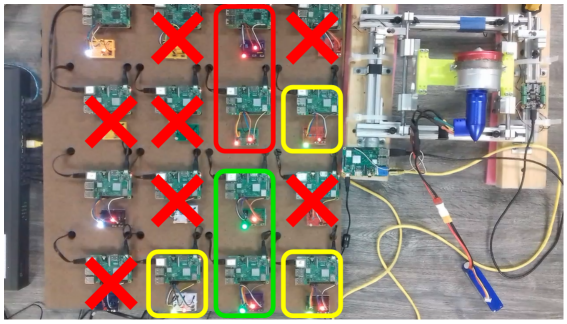
(a) After 4 faults, Application 3 has to be dropped. The isolated Tile cannot be used.



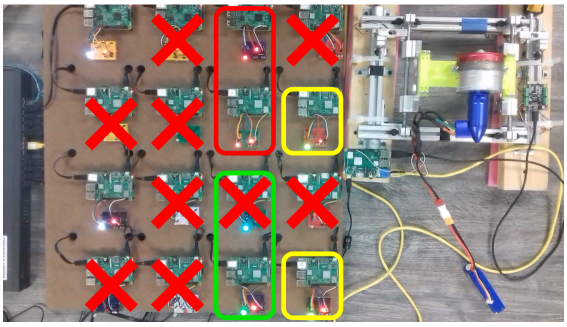
(b) When a Tile recovers from a fault, an allocation can be reallocated to it.



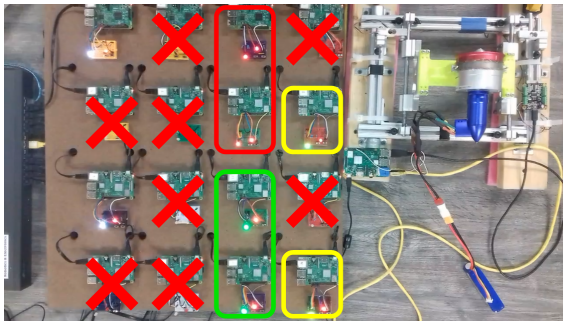
(c) After more faults, Application 3 and an allocator have to be dropped by lack of resources.



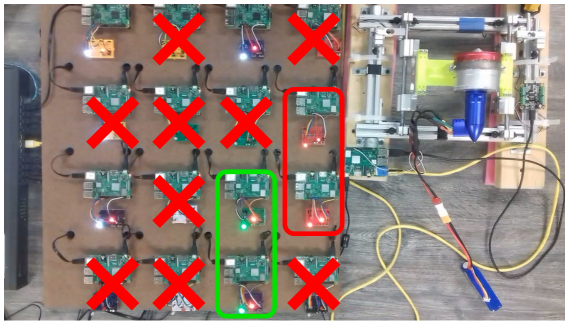
(d) One controller is dropped. The three allocators can still be executed, even if all first priority applications are not.



(e) One of the two controllers is affected by a computational fault: the fan stops since the fan voter does not trust any of the two incoherent controllers.



(f) The computational faults disappear: the fan starts again.



(g) All allocators have been dropped: no further reallocation is possible.

Figure 3.9: Result of the task allocation algorithm.
A full video of the demo is available at the link below.
https://gtvault-my.sharepoint.com/:v:/g/personal/lsutter6_gatech_edu/EZKi4kOwp19JlOgS4-Eo_lQBp3rQ-VCn8RnY879z8jPJ-Q?e=1QEb2y

CHAPTER 4
CONCLUSION

This work presented a decentralized allocation algorithm for parallel computing architectures, where individual Computational Units can be affected by faults. The described method consisted in representing the architecture by an abstract graph and formulating the allocation problem as an optimization problem, with the form of a Integer Linear Program.

Decentralizing the allocation process has been achieved through redundancy of the allocator executed on the architecture. That way, no centralized element decides of the allocation of the entire architecture.

An experimental reproduction of a multi-core architecture has also been built. It has been used to demonstrate the capabilities of the proposed allocation process to maintain operation of a physical system in a decentralized way while individual component fail.

The proposed work assumed that faults affecting the Computational Units of the architecture were automatically detected by the allocation algorithm, so that it is able to compute a new allocation every time a fault affects a Computational Unit. This work can be improved by defining a more precise model of the considered faults and a method to detect them. One first approach to identify dead Computational Unit would be a simple heartbeat that each CU would send to the allocators. A CU not sending its heartbeat would be considered faulty. One challenge to tackle in this approach is the fact that the allocators do not have a fixed position in the architecture, and therefore, the heartbeat of each CU would have to be broadcast through the entire architecture to be sure to reach all allocators. Another solution would be to include the position of the allocator in the allocation mes-

sage that they broadcast, so that the CU know where to send back their heartbeat. In both cases, the amount communication packets transmitted through the architecture drastically increases.

The second lead for improvement is the way communication isolation are taken into account in the allocation problem. For now, only individual nodes with all of their neighbors being faulty are considered isolated. However, an entire area of the architecture can be isolated from the allocator. The problem becomes quite tricky when the architecture is split in two halves that are isolated one from the other: a decision must be made to tell which area is isolated from the other. It seems that the area with the highest number of allocators should be privileged, since they are the ones that will send the new allocation to other CUs, and therefore the ones in the other isolated area will not be able to receive this new allocation. They should therefore be considered as lost CUs.

Also, it should be considered that not only the Computational Units can fails, but also the communication links between them. The effect of such faults would be the same as isolating the Computational Units from their neighbors and would make more of them unavailable. It would also change the communication paths usable to connect the allocators to other applications and would affect their position since minimizing these paths is a part of the optimization problem.

Appendices

APPENDIX A

COEFFICIENTS OF THE OBJECTIVE FUNCTION

This appendix provides the proof that the coefficients in the objective function from equation 2.3 allow to meet the requirements stated in Section 2.2.3. For convenience, this objective function is rewritten here:

$$\max \left\{ f(\boldsymbol{x}) = \sum_{k=1}^{N_{\text{apps}}} \alpha_k \cdot r_k - (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} M_j - \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} \left| X_{ij}^{\text{Comm},\, k} \right| \right\}, \tag{2.3}$$

where

$$\begin{aligned} \beta &= N_{\text{realloc}} \times N_{\text{CUs}} \times N_{\text{paths}}, \\ \alpha_{N_{\text{apps}}} &= (\beta + 1) \times N_{\text{nodes}} + \beta + 1, \\ \text{and } \forall k < N_{\text{apps}} : & \\ \alpha_k &= \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + (\beta + 1) \times N_{\text{nodes}} + \beta + 1. \end{aligned} \tag{2.4}$$

The requirements of Section 2.2.3 are also rewritten below.

1. When solving the optimization problem, the objective function 2.3 privileges executing any given application, even if it implies more reallocations and longer communication paths.

2. When solving the optimization problem, the objective function 2.3 privileges minimizing the number of reallocations, even if it implies longer communication paths.

3. When solving the optimization problem, the objective function 2.3 privileges executing a given application compared to running any number of applications with a lower priority.

The following theorems prove that these requirements are met.

Theorem 1. $\forall\ \tilde{k} \in [\![1,\ N_{apps}]\!]$:

$$\alpha_{\tilde{k}} > (\beta + 1) \sum_{j=1}^{N_{nodes}} 1 + \sum_{k=1}^{N_{realloc}} \sum_{j=1}^{N_{CUs}} \sum_{i=1}^{N_{paths}} 1,$$

that is, the contribution to the value of the objective function for executing application $\mathrm{app}_{\tilde{k}}$ *is greater than the maximum contribution for reducing the number of reallocations and the length of the communication paths.*

Proof. $\forall\ \tilde{k} \in [\![1,\ N_{\text{apps}}]\!]$:

$$\alpha_{\tilde{k}} \geq (\beta + 1) \times N_{\text{nodes}} + \beta,$$

by definition of $\alpha_{\tilde{k}}$.

Now,

$$(\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} 1 + \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} 1 = (\beta + 1) \times N_{\text{nodes}} + \beta.$$

So

$$\alpha_{\tilde{k}} > (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} 1 + \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} 1.$$

□

Theorem 1 proves that requirement 1 is met.

Theorem 2.

$$(\beta + 1) \times 1 > \sum_{k=1}^{N_{realloc}} \sum_{j=1}^{N_{CUs}} \sum_{i=1}^{N_{paths}} 1,$$

that is, the contribution to the value of the objective function for not reallocating one Application node is greater then the maximum contribution for reducing the length of communication paths.

Proof.

$$\beta + 1 > \beta = N_{\text{realloc}} \times N_{\text{CUs}} \times N_{\text{paths}} = \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} 1.$$

□

Theorem 2 proves that requirement 2 is met.

Theorem 3. $\forall\ \tilde{k} \in [\![1,\ N_{apps} - 1]\!]$:

$$\alpha_{\tilde{k}} > \sum_{l=\tilde{k}+1}^{N_{apps}} \alpha_l,$$

that is, the contribution to the value of the objective function for executing application $\text{app}_{\tilde{k}}$ *is greater than the contribution for executing every applications with lower priority than* $\text{app}_{\tilde{k}}$*, which are* $\text{app}_{\tilde{k}+1}$ *to* $\text{app}_{N_{apps}}$.

Proof. $\forall\ \tilde{k} \in [\![1,\ N_{\text{apps}}]\!]$:

$$\alpha_{\tilde{k}} = \sum_{l=\tilde{k}+1}^{N_{\text{apps}}} \alpha_l + (\beta + 1) \times N_{\text{nodes}} + \beta + 1 > \sum_{l=\tilde{k}+1}^{N_{\text{apps}}} \alpha_l,$$

since $(\beta + 1) \times N_{\text{nodes}} + \beta + 1 > 0$. □

Theorem 3 proves that requirement 3 is met.

REFERENCES

[1] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs—combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, 2012.

[2] N. Neves, N. Sebastião, D. Matos, P. Tomás, P. Flores, and N. Roma, "Multicore SIMD ASIP for next-generation sequencing and alignment biochip platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1287–1300, 2015.

[3] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallelization of min-cost flow for CAD applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1546–1557, 2010.

[4] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Dependable Computing Conference (EDCC), 2012 Ninth European*, IEEE, 2012, pp. 132–143.

[5] F. Reichenbach and A. Wold, "Multi-core technology–next evolution step in safety critical systems for industrial applications?" In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, IEEE, 2010, pp. 339–346.

[6] M. Alle, K. Varadarajan, A. Fell, C. R. Reddy, J. Nimmy, S. Das, P. Biswas, J. Chetia, A. Rao, S. K. Nandy, and R. Narayan, "REDEFINE: runtime reconfigurable polymorphic ASIC," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 2, 2009.

[7] L. M. Kinnan, "Use of multicore processors in avionics systems and its potential impact on implementation and certification," in *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, IEEE, 2009, 1–E.

[8] "Symmetric multi-processor arrangement, safety critical system, and method therefor," pat. US 2015/0254123 A1, Sep. 10, 2015.

[9] M. Oriol, T. Gamer, T. de Gooijer, M. Wahler, and E. Ferranti, "Fault-tolerant fault tolerance for component-based automation systems," in *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*, 2013, pp. 49–58.

[10] "GLPK reference manual," *GNU Linear Programming Kit*, 2012, https://www.gnu.org/software/glpk/TOCdocumentation.

[11] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*, Tenth. New York, NY, USA: McGraw-Hill, 2015.

[12] T. Guillaumet, E. Feron, P. Baufreton, F. Neumann, K. Madhu, M. Krishna, S. K. Nandy, R. Narayan, and C. Haldar, "Task allocation of safety-critical applications on reconfigurable multi-core architectures," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017, pp. 1–10.

[13] L. Sutter, T. Khamvilai, P. Monmousseau, J. B. Mains, E. Feron, P. Baufreton, F. Neumann, M. Krishna, S. K. Nandy, R. Narayan, and C. Haldar, "Experimental allocation of safety-critical applications on reconfigurable multi-core architecture," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, 2018, pp. 1–10.

[14] C. M. K. Israel Koren, *Fault tolerant systems*. Morgan Kaufmann Publishers, 2007, pp. 20–23.

[15] V. Rantala, T. Lehtonen, J. Plosila, *et al.*, *Network on chip routing algorithms*. Turku Centre for Computer Science, 2006.

[16] R. P. Foundation, *Raspberry Pi 3 Model B+*, https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/, Accessed June 2018.